A Term Rewrite System Framework for Code Carrying Theory

Paolo Picci

Abstract

Security is a fundamental aspect of every architecture based on a number of actors that exchange information among them, and the growing ubiquity of mobile and distributed systems has accentuated the problem. Mobile code is software that is transferred between systems and executed on a local system without explicit installation by the recipient, even if it is delivered through an insecure network or retrieved from an untrusted source. During delivery, the code may be corrupted or a malicious cracker could change the code damaging the entire system.

Code-Carrying Theory (CCT) is one of the technologies aimed at solving these problems. The key idea of CCT is based on proof-based program synthesis, where a set of axioms that define functions are provided by the code producer together with suitable proofs guaranteeing that defined functions obey certain requirements. The form of the function-defining axioms is such that it is easy to extract executable code from them. Thus, all that has to be transmitted from the producer to the consumer is a theory (a set of axioms and theorems) and a set of proofs of the theorems. There is no need to transmit code explicitly.

Many transformation systems for program optimization, program synthesis, and program specialization are based on fold/unfold transformations. We discuss a fold/unfold–based transformation framework for rewriting logic theories which is based on narrowing. When performing program transformation, we end up with a final program which is semantically equal to the initial one. We consider possibly non-confluent and non-terminating rewriting logic theories, and the rules for transforming these rewriting logic theories (definition introduction, definition elimination, folding, unfolding) that preserves the rewriting logic semantics of the original theory. The process for

obtaining a correct and efficient program can be split in two phases, which may be performed by different actors: the first phase is to write an initial maybe inefficient program whose correctness can be easily shown by hand or by automatic tools; in the second phase, the actor transforms the initial program by applying a certificate (a ordered set of instantiated rules of the framework) to derive a more efficient one.

The transformation system is naturally extended to CCT:

- Code Consumer provides the requirements to the code producer in the form of a rewrite theory.

- The Code Producer uses the fold/unfold-based transformation system in order to obtain an efficient implementation of the specified functions. Subsequently, the producer will send only a Certificate to be used by the Code Consumer to derive the program.

- Once the Certificate is received, the code consumer can apply the transformation sequence, described in the Certificate, to the initial theory, and the final program is automatically obtained. The strong correctness of the transformation system ensures that the obtained program is correct w.r.t. the initial Consumer specifications., so the Code Consumer does not need to check extra proofs provided by the Code Producer.

If we apply a correct certificate it is impossible to reach terms that carry malicious code or improper operations. Although the code producer should ensure the correctness of the certificate and the framework for fold/unfold transformation ensures the correctness and completeness of the final program, there is no guarantee that the requirements of the transformation rules are met correctly. During delivery, the certicate might be corrupted, or a malicious hacker might change the code. Potential problems can be categorized as security problems (i.e., unauthorized access to data or system resources), safety problems (i.e., illegal operations), or functional incorrectness (i.e., the delivered code fails to satisfy a required relation between its input and output) If there is no automatic support, it is very easy for an expert malicious hacker to intercept the certificate through an insecure network, modify it and resend to the code consumer. We have demonstrated that we cannot apply a certificate regardless of its contents. We need to check that all the operation descriptions to be carried over to the system are lawful and all operations

are done in the correct order.

We implemented in a prototypical system, the transformation framework extended with the infrastructure for certificate checking, which consists of a suit of tools. The implementation is written in Maude easily using the reflection, Python and some scripts in Bash.
So the Code Consumer which uses our framework can receive, check and apply a certificate to a initial theory; detect and refuse bad certificates with a detailed report; and then avoids data corruption or attacks from malicious actors.